John von Neumann Institute for Computing

Parallel Computation of Stochastic Groundwater Flow

Andreas Keese, Hermann G. Matthies

published in

NIC Symposium 2004, Proceedings,
Dietrich Wolf, Gernot Münster, Manfred Kremer (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **20**, ISBN 3-00-012372-5, pp. 399-408, 2003.

http://www.fz-juelich.de/nic-series/volume20

Parallel Computation of Stochastic Groundwater Flow

Andreas Keese and Hermann G. Matthies

Institute of Scientific Computing, Technical University Braunschweig
Hans-Sommer-Strasse 65, 38106 Braunschweig, Germany
E-mail: wir@tu-bs.de

If a stochastic model is used to describe uncertainties, the physical system may be described by a stochastic partial differential equation (SPDE). A discretisation by a Galerkin ansatz with tensor-products of finite element functions and stochastic ansatz functions yields a large system of equations that can be efficiently solved by iterative methods. Due to its sheer size, parallel techniques are required, and we have implemented a "hierarchical parallel solver" for this: our solver uses a (possibly parallel) deterministic solver for the spatial discretisation. Coarser grained levels of parallelism are implemented by distributing the unknowns over the processors and by running different instances of the (possibly parallel) deterministic solver in parallel.

1 Motivation

Uncertainties remain in all models of the physical reality, and the quality of numerical prognoses is limited by the information available about the system. If prognoses are computed with an accuracy higher than merited by the available information, then the computing power at hand might be put to use more effectively by computing quantitative estimates for the uncertainties in the response.

Soil properties are very hard to measure. Hence, the material parameters used in the simulation of groundwater flows are usually flawed by uncertainties. Apart from homogenisation techniques, such uncertainties have been accounted for by stochastic models[1,2]. The system is then described by a stochastic partial differential equation (SPDE)[3–6] and may be solved by stochastic finite element methods (SFEM)[5–10].

2 Stochastic Partial Differential Equations

As a simple example, we consider a groundwater flow problem in a region $R \subset \mathbb{R}^d$ (see Fig. 1), where the flux is related to the hydraulic head gradient by Darcy's law. We may model uncertainties in the soil by describing the hydraulic conductivity κ by random variables $\kappa(x,\omega), x \in R$ on a probability space Ω, where $\omega \in \Omega$ denotes an elementary event. Consequently, $\kappa : R \times \Omega \to \mathbb{R}$ is a random field[11,12,2]; see Fig. 1. The hydraulic head is then also a random field $u(x,\omega)$, and it satisfies the elliptic SPDE

$$-\nabla \cdot (\kappa(x,\omega)\nabla u(x,\omega)) = f(x,\omega). \tag{1}$$

Here $f(x,\omega)$ is a random field modelling the sources and sinks, and we assume that appropriate boundary conditions are imposed.

The properties of interest are statistics of the hydraulic head, like its mean, its variance, or its probability to exceed some threshold; see Fig 2.

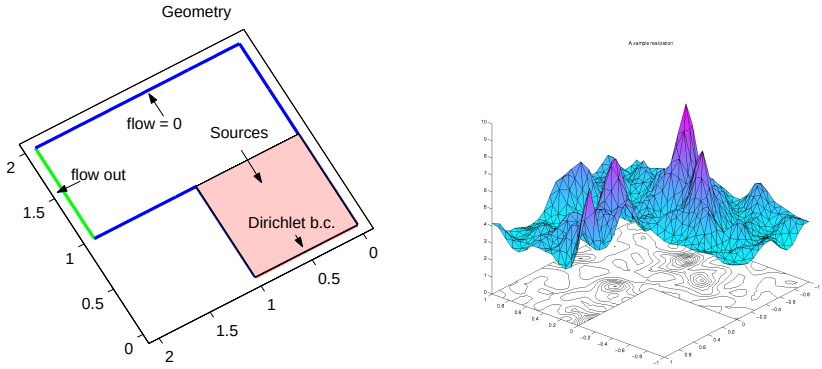

Figure 1. Geometry and a realisation of the hydraulic conductivity.

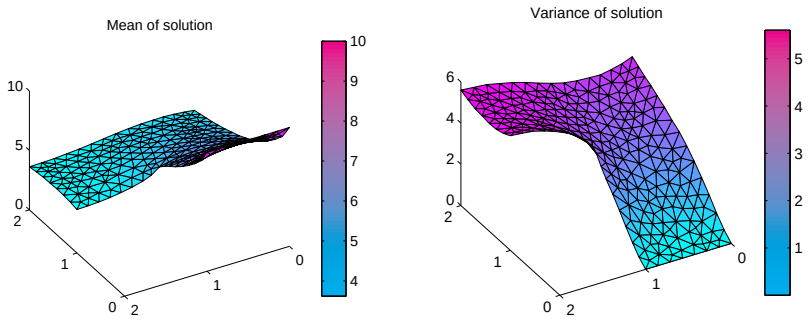

Figure 2. Mean and variance of solution.

3 Discretisation

Such statistics may be computed by approximating all random quantities in a finite number of independent random variables and then integrating over the appropriate probability space (taking the expectation). Monte Carlo integration may be used for this, but the effort of performing Monte Carlo simulations is high, and hence alternative techniques have been developed, which may be advantageous if the number of independent random variables is not too high or if the variance of the answer is large[7].

We use the *spectral stochastic finite element method*[13]: The random fields are represented by their Karhunen-Loève expansion[11], and the solution is obtained by an ansatz $u(x,\omega) = \sum_{i=1}^{n} \sum_{\beta=1}^{k} N_i(x) H_\beta(\omega) u_i^{(\beta)}$ of tensor-products of finite element shape functions $N_1(x), \ldots, N_n(x),\ x \in R$, and stochastic functions $H_1(\omega), \ldots, H_k(\omega),\ \omega \in \Omega$, that are chosen here as multivariate Hermite polynomials (the so-called polynomial chaos). The unknowns may be written as a block vector $\mathbf{u} = (\ldots, \boldsymbol{u}^{(\beta)}, \ldots)^T$ consisting of subvectors $\boldsymbol{u}^{(\beta)}$, each of which is a coefficient vector of the spatially discretised problem.

Application of Galerkin conditions yields a structured system of block-equations that

can be written[5,14] for the linear problem as

$$\mathbf{Ku} = \left[\sum_i \sum_\gamma \kappa_i^{(\gamma)} \Delta^{(\gamma)} \otimes \mathbf{K}_i \right] \mathbf{u} = \mathbf{f}. \tag{2}$$

The matrices $\mathbf{K}_i$ correspond to calling the deterministic solver with special material parameters (with the Karhunen-Loève eigenfunctions). The matrices $\Delta^{(\alpha)}$ contain the contributions of the stochastic ansatz, and their size is equal to the number of stochastic ansatz functions. The scalars $\kappa_i^{(\alpha)}$ are computed from the statistics of the stochastic materials, and the block vector **f** is computed from the sinks, sources, and boundary conditions.

Due to the tensor-product-structure, the number of equations is large—it is the size of the spatial times the size of the stochastic ansatz. To solve this large system of equations, we may exploit its special structure, and efficient solvers based on preconditioned Krylov-methods and multilevel-solvers in the stochastic dimension have been implemented[15,16]. The solvers call existing software in a black-box fashion to compute the matrix vector products $\mathbf{K}_i \boldsymbol{u}^{(\beta)}$, and they use block-diagonal preconditioners that are based on applying the existing deterministic solver to each subvector $\boldsymbol{u}^{(\beta)}$.

4 Parallelisation

In order to simulate realistic problems[17], large ansatz-spaces are necessary, and hence the solver was parallelised. As we use Krylov-type solvers, the parallelisation was performed by parallelising the block-matrix-vector product $\mathbf{v} := \mathbf{Ku}$ and the preconditioner.

According to Eq. (2), the block-matrix-vector product is computed as

$$\boldsymbol{v}^{(\alpha)} = (\mathbf{Ku})_\alpha = \sum_i \sum_\beta \sum_\gamma \sqrt{\lambda_i} \xi_i^{(\gamma)} \Delta_{\alpha,\beta}^{(\gamma)} \mathbf{K}_i \, \boldsymbol{u}^{(\beta)}. \tag{3}$$

The parallelisation was performed in a configurable and hierarchical manner:

Parallel Deterministic Solver: If the deterministic solver is a parallel program, e.g. based on a domain decomposition of the spatial region, then each matrix $\mathbf{K}_i$ is distributed over a set of processors, and the vectors $\boldsymbol{u}^{(\beta)}$ and $\boldsymbol{v}^{(\alpha)}$ are distributed accordingly. We divide the available processors into N_{pg} equally sized processor groups ("p-groups"). These are our smallest building blocks for the coarser levels of the parallelisation. Each p-group runs one instance of the deterministic solver and stores parts of the block-vectors **u** and **v**. Here, each subvector $\boldsymbol{u}^{(\beta)}$ and $\boldsymbol{v}^{(\alpha)}$ is distributed over the processors in the group as required by the deterministic solver.

What instances of the deterministic solver are run on which p-group, and what parts of the block vectors **u** and **v** are stored on which p-group, depends on how the other levels of the parallelisation are configured.

Parallel Execution of Deterministic Solvers: We allow to simultaneously run different instances of the deterministic solver, i.e. to execute the sum over i in Eq. (3) in parallel.

We may store more than one $\mathbf{K}_i$ on every p-group, and in this case it is necessary to exchange the material parameter in the deterministic solver while executing the block-matrix-vector product. For this we may either call an appropriate function of the deterministic solver or restart it with another material parameter.

For a faster performance, we may replicate the matrices $\mathbf{K}_i$, i.e. run identical instances simultaneously on several p-groups. We characterise their distribution over the p-groups by the number N_K of replications: each of the matrices $\mathbf{K}_i$ (or of the instances of the solver) is copied to N_K different p-groups.

Distribution of Block Vectors: To allow large ansatz spaces, we distribute **u** and **v** over the p-groups. This amounts to parallelising both the sum over β and the loop over α.

If the number of unknowns is not too large, it may be favorable to store each vector more than once. This increases memory requirements but reduces execution times.

We characterise the distribution of the block vectors over the p-groups by the number N_V of their replications—copies of each subvector $\boldsymbol{u}^{(\beta)}$ and $\boldsymbol{v}^{(\alpha)}$ are stored on N_V different p-groups.

4.1 Distribution of Data

To highlight parallelisation aspects, let us discuss some examples. For simplicity, we assume here that four processor groups are available, that four matrices $\mathbf{K}_0, \ldots, \mathbf{K}_3$ are to be distributed and that each block vector comprises 12 subvectors. Of course, realistic examples use more matrices and longer block-vectors, but this example suffices for demonstration purposes.

If the vectors $\boldsymbol{u}^{(\beta)}$ or $\boldsymbol{v}^{(\alpha)}$ or the matrices $\mathbf{K}_i$ are stored in a distributed manner, then the p-groups need to exchange data while executing the block-matrix-vector product. Each matrix $\mathbf{K}_i$ may be identified with an instance of the deterministic solver, hence we do not exchange them.

Instead, we exchange parts of the block-vectors $\boldsymbol{u}^{(\beta)}$ and $\boldsymbol{v}^{(\alpha)}$. Before and after executing the matrix-vector product, all block-vectors are distributed in the same fashion. While executing the block-matrix-vector product, they are exchanged between the p-groups. All communications may be performed as cyclical shifts across subsets of the p-groups (the first p-group in the set sends its part of the block-vector to the second, the second sends to the third, and so on, and the last p-group sends its part to the first p-group).

EXAMPLE 1: The most efficient distribution of data in terms of memory demands results if every matrix and every vector is stored only once:

p-group	$\mathbf{K}_i$	**u**	**v**
pg_1	$\mathbf{K}_0$	$\boldsymbol{u}^{(1)} \cdots \boldsymbol{u}^{(3)}$	$\boldsymbol{v}^{(1)} \cdots \boldsymbol{v}^{(3)}$
pg_2	$\mathbf{K}_1$	$\boldsymbol{u}^{(4)} \cdots \boldsymbol{u}^{(6)}$	$\boldsymbol{v}^{(4)} \cdots \boldsymbol{v}^{(6)}$
pg_3	$\mathbf{K}_2$	$\boldsymbol{u}^{(7)} \cdots \boldsymbol{u}^{(9)}$	$\boldsymbol{v}^{(7)} \cdots \boldsymbol{v}^{(9)}$
pg_4	$\mathbf{K}_3$	$\boldsymbol{u}^{(10)} \cdots \boldsymbol{u}^{(12)}$	$\boldsymbol{v}^{(10)} \cdots \boldsymbol{v}^{(12)}$

Table 1. Example for parallel matrix-vector-product, most efficient memory usage.

We need to add every possible product $\mathbf{K}_i \boldsymbol{u}^{(\beta)}$ to every $\boldsymbol{v}^{(\alpha)}$. In this example, all $\boldsymbol{u}^{(\beta)}$ must thus be cyclically shifted across all processor groups, and for every configuration of **u** we need to shift all $\boldsymbol{v}^{(\alpha)}$ across all p-groups. For every configuration of **v** and **u** we add the local contributions to the right hand side.

If there is no redundancy, as in this example, then in total N_{pg} cyclical shifts of **u** and N_{pg}^2 cyclical shifts of **v** are required. The memory demands scale well with the number of p-groups, but the number of required cyclical block-vector shifts grows quadratically with

the number of p-groups. As Fig. 3 shows, there may be an optimum number of p-groups for a given problem size, and adding more p-groups may result in an increased total execution time.

To speed up the block-matrix-vector product, we may introduce redundancy. First we consider a replication of the matrices $\mathbf{K}_i$. We need to support this type of redundancy anyway, as the number of matrices $\mathbf{K}_i$ may be smaller than the number of available p-groups.

EXAMPLE 2: We now store each $\mathbf{K}_i$ twice and the block vector once:

p-group	$\mathbf{K}_i$		**u**	**v**
pg_1	$\mathbf{K}_0$	$\mathbf{K}_1$	$\boldsymbol{u}^{(1)} \cdots \boldsymbol{u}^{(3)}$	$\boldsymbol{v}^{(1)} \cdots \boldsymbol{v}^{(3)}$
pg_2	$\mathbf{K}_2$	$\mathbf{K}_3$	$\boldsymbol{u}^{(4)} \cdots \boldsymbol{u}^{(6)}$	$\boldsymbol{v}^{(4)} \cdots \boldsymbol{v}^{(6)}$
pg_3	$\mathbf{K}_0$	$\mathbf{K}_1$	$\boldsymbol{u}^{(7)} \cdots \boldsymbol{u}^{(9)}$	$\boldsymbol{v}^{(7)} \cdots \boldsymbol{v}^{(9)}$
pg_4	$\mathbf{K}_2$	$\mathbf{K}_3$	$\boldsymbol{u}^{(10)} \cdots \boldsymbol{u}^{(12)}$	$\boldsymbol{v}^{(10)} \cdots \boldsymbol{v}^{(12)}$

Table 2. Example for parallel matrix-vector-product, operators replicated.

Again, we need a cyclic shift of **u** across all p-groups. But in contrast to example 1, we need to exchange the right hand side only between those processor groups that store a complete set of matrices $\mathbf{K}_i$, i.e. it is sufficient to exchange the $\boldsymbol{v}^{(\alpha)}$ between pg_1 and pg_2 and between pg_3 and pg_4.

If there are N_K copies of every $\mathbf{K}_i$, we need to perform N_{pg} cyclical shifts of **u** and N_{pg}^2/N_K shifts of **v**. This distribution still scales well with the number of processor groups in terms of memory demands but requires less communications than example 1.

With even more redundancy, every processor group stores all matrices $\mathbf{K}_i$ and hence the complete block matrix. Then **u** must again be cyclically exchanged between all p-groups, but the right hand side needs not to be exchanged. The number of cyclical block-vector shifts then grows linearly with the numbers of processors; see the speedup-measurements in Fig. 3.

The costs of exchanging the material parameter in a p-group may be high, e.g. if this requires a restart of the deterministic solver. It may then be advantageous to assign to each p-group only one deterministic solver instance. To speed up the solution process, we may then hold more than one copy of each block vector.

EXAMPLE 3: The following memory distributions demonstrate this:

p-group	$\mathbf{K}_i$	**u**	**v**
pg_1	$\mathbf{K}_0$	$\boldsymbol{u}^{(1)} \cdots \boldsymbol{u}^{(6)}$	$\boldsymbol{v}^{(1)} \cdots \boldsymbol{v}^{(6)}$
pg_2	$\mathbf{K}_1$	$\boldsymbol{u}^{(7)} \cdots \boldsymbol{u}^{(12)}$	$\boldsymbol{v}^{(7)} \cdots \boldsymbol{v}^{(12)}$
pg_3	$\mathbf{K}_2$	$\boldsymbol{u}^{(1)} \cdots \boldsymbol{u}^{(6)}$	$\boldsymbol{v}^{(1)} \cdots \boldsymbol{v}^{(6)}$
pg_4	$\mathbf{K}_3$	$\boldsymbol{u}^{(7)} \cdots \boldsymbol{u}^{(12)}$	$\boldsymbol{v}^{(7)} \cdots \boldsymbol{v}^{(12)}$

Table 3. No redundancy in operator, block-vectors stored twice.

If the block-vectors are stored twice, then **u** must be exchanged within each set of processor groups storing a complete block-vector. This requires $N_{pg}/2$ cyclical exchanges of **u**. After all the contributions $\mathbf{K}_i\boldsymbol{u}^{(\beta)}$ have been added to the right hand side, a parallel

prefix operation is required: the $\boldsymbol{v}^{(\alpha)}$ on pg_1 and pg_2 need to be added to their counterparts on pg_3 and pg_4, and the result of the sum needs to be redistributed (in MPI-terminology this is an "Allreduce"-operation).

With more redundance, even less communication is required:

p-group	$\mathbf{K}_i$	**u**	**v**
pg_1	$\mathbf{K}_0$	$\boldsymbol{u}^{(1)} \dots \boldsymbol{u}^{(12)}$	$\boldsymbol{v}^{(1)} \dots \boldsymbol{v}^{(12)}$
pg_2	$\mathbf{K}_1$	$\boldsymbol{u}^{(1)} \dots \boldsymbol{u}^{(12)}$	$\boldsymbol{v}^{(1)} \dots \boldsymbol{v}^{(12)}$
pg_3	$\mathbf{K}_2$	$\boldsymbol{u}^{(1)} \dots \boldsymbol{u}^{(12)}$	$\boldsymbol{v}^{(1)} \dots \boldsymbol{v}^{(12)}$
pg_4	$\mathbf{K}_3$	$\boldsymbol{u}^{(1)} \dots \boldsymbol{u}^{(12)}$	$\boldsymbol{v}^{(1)} \dots \boldsymbol{v}^{(12)}$

Table 4. No redundancy in operator, highest redundancy in block-vectors.

The only communication required is now a collective sum of the right hand side with a redistribution. Accordingly, the measurements in Fig. 4 show almost perfect efficiency.

Highest redundancy in the block-vectors as in example 4 leads to almost perfect speed-up, but it does not scale in terms of memory demands. However, it is an extreme case, and we allow to combine both types of redundancy, so that the parallel program may be configured with respect to the problem of interest.

4.2 Parallel Block-Matrix-Vector Product, Algorithm

Every processor executes the pseudocode shown in algorithm 1 to perform the parallel block-matrix-vector product.

In the algorithm, each p-group is a set of processors running the deterministic solver. We require that the spatial degrees of freedom are arranged in the same manner on each processor group, and we denote the part of a vector locally stored on the processor executing the algorithm by *local_dof*, i.e. $\boldsymbol{v}^{(\alpha)}(\textit{local\_dof})$ is the local part of $\boldsymbol{v}^{(\alpha)}$.

We hold N_K copies of the operator, where N_K is a divisor of the number N_{pg} of p-groups: every matrix $\mathbf{K}_i$ is copied onto N_K p-groups and distributed inside each of them according to the arrangement of the spatial degrees of freedom. By *local_matrices* we denote all $\mathbf{K}_i$ stored on the local p-group. As the matrices $\Delta^{(\gamma)}$ are very sparse, they are stored as a sorted list of non-zero entries on every processor. If a matrix $\mathbf{K}_i$ is stored on a processor, this processor also holds a copy of all the corresponding $\xi_i^{(\gamma)}$. Altogether there are thus N_K sets of processor groups that store the complete operator. Each of these sets may apply the whole block-matrix-vector product to the local parts of the block vector **u** and add it to the local parts of the right hand side. We denote these sets of p-groups as "operator groups". For example, Table 2 contains the two operator groups $\{pg_1, pg_2\}$ and $\{pg_3, pg_4\}$.

We store N_{vec} copies of each block-vector **u** and **v**, where N_{vec} is a divisor of the number of p-groups, and where either N_{vec} is a divisor of N_K, or N_K is a divisor of N_{vec}. As a result, we have N_{vec} sets of $vgsize := N_{pg}/N_{vec}$ p-groups, each storing a complete copy of every block-vector. We call these sets of p-groups "vector groups", and we denote the local parts of the vectors **u** and **v** by $\textit{local}_{\mathbf{u}}$ and $\textit{local}_{\mathbf{v}}$. For example, in Table 3 appear the two vector groups $\{pg_1, pg_2\}$ and $\{pg_3, pg_4\}$ and in Table 4 every processor group is a vector group.

Algorithm 1 Parallel Block-Matrix-Vector Product.

for all $\alpha \in$ *local*$_{\mathbf{v}}$ **do**
  $\boldsymbol{v}^{(\alpha)}$(*local_dof*) $\leftarrow 0$
end for
for all $i \in$ *local_matrices* **do**
  Activate deterministic solver $\mathbf{K}_i$ (set material)
  for $vgsize$ times **do** {cycle **u**}
    for all $\beta \in$ *local*$_{\mathbf{u}}$ **do**
      perform collective operation in processor group
        $\boldsymbol{w} := \mathbf{K}_i \boldsymbol{u}^{(\beta)}$
      end collective operation
      for $vgsize/N_K$ times **do**
        for all $\alpha \in$ *local*$_{\mathbf{v}}$ **do**
          $c_{i,\beta,\alpha} := \sum_\gamma \sqrt{\lambda_i}\xi_i^{(\gamma)}\Delta_{\alpha,\beta}^{(\gamma)}$
          $\boldsymbol{v}^{(\alpha)}$(*local_dof*) $\leftarrow c_{i,\beta,\alpha}\boldsymbol{w}$(*local_dof*) $+ \boldsymbol{v}^{(\alpha)}$(*local_dof*)
        end for
        perform collective operation in operator group
          exchange **v** cyclically inside operator group
          obtain new *local*$_{\mathbf{v}}$
        end collective operation
      end for
    end for
    perform collective operation in intersection of vector & matrix group
      exchange **u** cyclically in intersection of vector & matrix group
      obtain new *local*$_{\mathbf{u}}$
    end collective operation
  end for
end for
perform collective operation in all processors {MPI-Allreduce operation}
    sum up all $\boldsymbol{v}^{(\alpha)}$ over all vector groups
    distribute result of summation to all $\boldsymbol{v}^{(\alpha)}$ in all vector groups
end collective operation

4.3 Parallel Solver

The parallel solver is implemented by a conjugate gradients algorithm, which is parallelised by calling the parallel matrix-vector product. As preconditioner we use the block-diagonal preconditioner discussed above by running the (maybe parallel) deterministic solver on the respective processor groups.

The parallelisation of the block-preconditioner is straightforward: We apply the deterministic solver with the mean as material to the local subvectors of a block-vector. If the block-vectors are replicated, we take care that every component of the result is computed on only one processor group during the preconditioning stage and, if necessary, we exchange the computed components of the result between the processor groups after the block-preconditioning stage is finished.

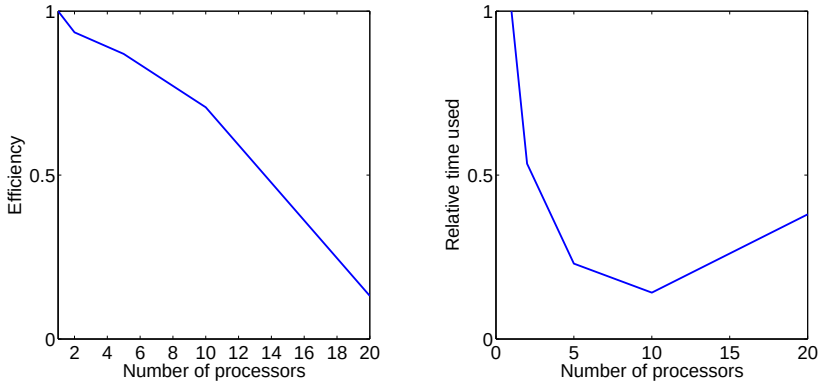

Figure 3. Speedup Measurements, no redundancy. **Left plot**: Parallel efficiency. **Right plot**: relative execution time.

5 Speedup-Measurements

The parallel solver was written in C++ using the portable communication library MPI. For the spatial discretisation we imported here matrices from an external FEM-code and used a parallel matrix-vector product and linear solver built with PetSC[18]. The timings were performed on a Cray T3E at the Research Centre Jülich.

The speedup-measurements that are shown in Fig. 3 and Fig. 4 are *preliminary results* obtained for a fixed problem size on one to twenty processors. Computations with more processors, where the problem size is scaled with the number of processors, are in work and will be published elsewhere.

The number of iterations required by the solver grows with the number of unknowns; to measure only the speedup, the problem size was therefore kept constant, and we used

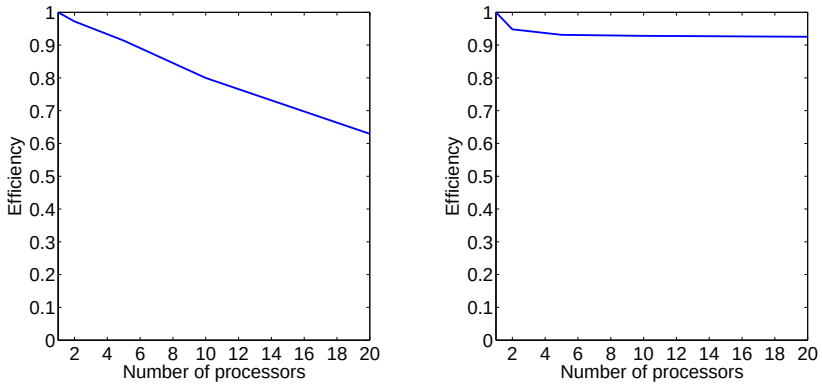

Figure 4. Parallel efficiency with redundancy. **Left**: Every processor stores all matrices, block-vectors stored redundance-free. **Right**: Matrices stored without redundance, every processor stores complete block vectors.

a small problem to make it solvable on one processor. The domain shown in Fig. 1 was discretised with 750 spatial degrees of freedom and 1,540 stochastic degrees of freedom (in total $1,155,500$ equations), twenty operators $\mathbf{K}_1, \ldots, \mathbf{K}_{20}$ were used. They were run as sequential matrix-vector multiplications (each p-group comprised only one processor).

The measurements in Fig. 3 show the parallel efficiency and relative execution times with no redundancies in storing the operators or block-vectors. As discussed in example 1, the number of cyclical vector shifts increases quadratically with the number of p-groups, and as the right plot shows, there is an optimum number of processors in terms of total computing time. Increasing the number of processors beyond this optimum number results in an increase of the total execution time.

The left plot in Fig. 4 shows the parallel efficieny if all matrices $\mathbf{K}_1, \ldots, \mathbf{K}_{20}$ are stored on every processor. As discussed in example 2, the number of cyclical vector shifts grows linearly with the number of p-groups. The right plot in Fig. 4 shows the efficiency if every p-group stores all block-vectors. The speedup is then almost perfect as the only parallel communication in every iteration is a parallel prefix operation and a redistribution of the data at the end of the matrix-vector product.

6 Conclusions

The parallelisation was performed on a hierarchy of different levels, allowing to configure the parallelisation to match the problem at hand. By introducing redundancies, execution time may be traded for memory demands, and good speedups were obtained. Speedup measurements on more processors, where the problem size is scaled with the number of processors, are in preparation. Better speedup is expected if the problem size is scaled with the number of processors as the relation of computation to communication becomes more favorable. Also, measurements where the deterministic solver is executed in parallel are ongoing and will be published elsewhere.

The example presented here looks simple, but the stochastic uncertainties increase its complexity considerably. The solver may integrate existing codes for the spatial discretisation and hence may be applied to more complex problems. Our goal is to implement a general-purpose version of the stochastic finite element method that may be applied to real-life problems, and as the resulting systems of equations are large, the parallel solver presented here is an important step towards this goal.

Acknowledgements

The admission to the Cray T3E of the John von Neumann Institute for Computing at the Research Centre Jülich is gratefully acknowledged.

References

1. G. Dagan and S. P. Neuman, editors. *Subsurface Flow and Transport: A stochastic Approach*. Cambridge University Press, Cambridge, 1997.
2. George Christakos. *Random field models in earth sciences*. Academic Press, New York, NY, 1992.

3. Helge Holden, Bernt Øksendal, Jan Ubøe, and Tu-Sheng Zhang. *Stochastic Partial Differential Equations*. Birkhäuser, Basel, 1996.
4. Ivo Babuška and Panagiotis Chatzipantelidis. On solving linear elliptic stochastic partial differential equations. *Computer Methods in Applied Mechanics and Engineering*, 191:4093–4122, 2002.
5. Hermann G. Matthies and Andreas Keese. Galerkin methods for linear and nonlinear elliptic stochastic partial differential equations. submitted to the *Computer Methods in Applied Mechanics and Engineering*, 2003.
6. Andreas Keese. A review of recent developments in the numerical solution of stochastic PDEs (stochastic finite elements). Informatikbericht 2003-6, Technische Universitat Braunschweig, Braunschweig, 2003.
7. Andreas Keese and Hermann G. Matthies. Numerical methods and Smolyak quadrature for nonlinear stochastic partial differential equations. *submitted to the SIAM Journal of Scientific Computing*, 2003.
8. Hermann G. Matthies, Christoph E. Brenner, Christoph G. Bucher, and C. Guedes Soares. Uncertainties in probabilistic numerical analysis of structures and solids—stochastic finite elements. *Structural Safety*, 19(3):283–336, 1997.
9. Gerhart I. Schuëller. A state-of-the-art report on computational stochastic mechanics. *Probabilistic Engineering Mechanics*, 14(4):197–321, 1997.
10. Bruno Sudret and Armen Der Kiureghian. Stochastic finite element methods and reliability. A state-of-the-art-report. Technical Report UCB/SEMM-2000/08, University of California, Berkeley, CA, 2000.
11. Robert J. Adler. *The Geometry of Random Fields*. John Wiley & Sons, Chichester, 1981.
12. Erik Vanmarcke. *Random Fields: Analysis and Synthesis*. The MIT Press, Cambridge, MA, 3^{rd} edition, 1988.
13. Roger Ghanem and Pol Spanos. *Stochastic finite elements—A spectral approach*. Springer, Berlin, 1991.
14. Andreas Keese and Hermann G. Matthies. Hierarchical parallel solution of stochastic systems. In K. J. Bathe, editor, *Computational Fluid and Solid Mechanics 2003*, volume 2, pages 2023–2025. Elsevier, Amsterdam, 2003.
15. Hermann G. Matthies and Andreas Keese. Multilevel methods for stochastic systems. In *ECCM-2001, Proceedings of the Second European Conference on Computational Mechanics*, Cracow, Poland, 2001.
16. Hermann G. Matthies and Andreas Keese. Multilevel solvers for the analysis of stochastic systems. In K.J. Bathe, editor, *Computational Fluid and Solid Mechanics*, pages 1620–1622. Elsevier, Amsterdam, 2001.
17. Volker Schulz, Andreas Bardossy, and Rainer Helmig. Conditional statistical inverse modeling in groundwater flow by multigrid methods. *Computational Geosciences*, 3:49–68, 1999.
18. Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202, Basel, 1997. Birkhäuser.